CSC 8400: Bit Manipulation Lab

Introduction

The purpose of this lab is to become more familiar with bit-level representations and manipulations. You'll do this by solving a series of programming "puzzles." Many of these puzzles are quite artificial, but you'll find yourself thinking much more about bits in working your way through them. You may work in groups of two on this lab. If you do not have a partner and would like one, please let me know.

Our Linux system

This lab must be completed on a Linux machine. Our Unix cluster has two Linux machines named `felix` and `helix`. You can log either of these machines either directly using SSH, or through `csgate` using ssh – log into `csgate` first, then at the shell prompt type in

```
ssh username@felix.csc.villanova.edu
```

Hand Out Instructions

Step 1. Make sure that you complete this lab on `helix` or `felix`. Start by creating a directory `datalab` in your `csc8400` directory.

Step 2. Copy all files from `/mnt/a/mdamian/courses/csc8400/datalab` into your `datalab` directory using the command

```
cp /mnt/a/mdamian/courses/csc8400/datalab/*  ~/csc8400/datalab
```

This will cause ten files to be copied in your `datalab` directory.

The only file you will be modifying and turning in is `bits.c`.

The file `btest.c` allows you to evaluate the functional correctness of your code. The file `README` contains additional documentation about `btest`. The file `dlc` is a compiler binary that you can use to check your solutions for compliance with the coding rules. The remaining files are used to build `btest`.

Step 3. Use the command `make` to compile the project, then type `btest` to run it. *Repeat this step each time you make a modification to your* `bits.c` *file and want to test it.*

Step 4. Looking at the file `bits.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. Do this right away so you don't forget. Also insert your team name in the first line in your `Makefile`.

The file `bits.c` contains skeletons for 12 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are *only* allowed to use the following eight operators:

```
! ~ & ^ | + << >>
```

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than 8 bits. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

Step 5 (final). Enter a fun and friendly competition against the instructor. One goal of this lab is to solve each data puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for each puzzle are winners. Instructions for entering the contest are available online at

`http://www.csc.villanova.edu/~mdamian/csc8400/dlcontest.html`

Students submit their entries by running the script called `submit.pl`. This sends mail to a spool file, which is continuously scanned by a daemon that summarizes the results on this website.

Evaluation

Your code will be compiled with GCC and run and tested on felix. Your score will be computed out of a maximum of 90 points based on the following distribution:

60 Correctness of code running on the `felix` machine.

24 Performance of code, based on number of operators used in each function.

6 Style points, based on the quality of your solutions and your comments.

The 12 puzzles have been given a difficulty rating between 1 and 4. I will evaluate your functions using `btest`. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, half credit if it fails one test, and no credit otherwise.

Regarding performance, our main concern at this point in the course is that you can get the right answer. However, we want to instill in you a sense of keeping things as short and simple as you can. Furthermore, some of the puzzles can be solved by brute force, but we want you to be more clever. Thus, for each function we've established a maximum number of operators that you are allowed to use for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions.

Finally, we've reserved 6 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

Name	Description	Rating	Max Ops
`bitNor(x,y)`	`~(x|y)` using only `&` and `~`	1	8
`bitXor(x,y)`	`^` using only `&` and `~`	2	14
`isNotEqual(x,y)`	`x != y?`	2	6
`getByte(x,n)`	Extract byte `n` from `x`	2	6
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`logicalShift(x,n)`	Logical right shift `x` by `n`	3	16
`bang(x)`	Compute `!x` without using `!`	4	12
`leastBitPos(x)`	Mark least significant 1 bit	4	30

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`tmax(void)`	largest two's complement integer	1	4
`isNonNegative(x)`	`x >= 0?`	3	6
`abs(x)`	absolute value	4	10
`addOK(x,y)`	Does `x+y` overflow?	3	20

Table 2: Arithmetic Functions

Part I: Bit manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. The "Rating" field gives the difficulty rating (the number of points) for the puzzle, and the "Max ops" field gives the maximum number of operators you are allowed to use to implement each function.

Function `bitNor` computes the NOR function. That is, when applied to arguments `x` and `y`, it returns `~(x|y)`. You may only use the operators `&` and `~`. Function `bitXor` should duplicate the behavior of the bit operation `^`, using only the operations `&` and `~`.

Function `isNotEqual` compares `x` to `y` for inequality. As with all *predicate* operations, it should return `1` if the tested condition holds and `0` otherwise.

Function `getByte` extracts a byte from a word. The bytes within a word are ordered from 0 (least significant) to 3 (most significant). Function `copyLSB` replicates a copy of the least significant bit in all 32 bits of the result. Function `logicalShift` performs logical right shifts. You may assume the shift amount n satisfies $1 \leq n \leq 31$.

Function `bang` computes logical negation without using the `!` operator. Function `leastBitPos` generates a mask consisting of a single bit marking the position of the least significant one bit in the argument. If the argument equals 0, it returns 0.

Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Function `tmax` returns the largest integer.

Function `isNonNegative` determines whether `x` is less than or equal to 0.

Function `abs` is equivalent to the expression `x<0?-x:x`, giving the absolute value of `x` for all values other than `TMin` (the smallest integer value, which is negative).

Function `addOK` determines whether its two arguments can be added together without overflow.

Advice

- You are welcome to do your code development using any system or compiler you choose. Just make sure that the version you turn in compiles and runs correctly on our Linux machines (helix or felix). If it doesnt compile, I can't grade it.
- The `dlc` program, a modified version of an ANSI C compiler, will be used to check your programs for compliance with the coding style rules. The typical usage is

  ```
  ./dlc bits.c
  ```

  Type `./dlc -help` for a list of command line options. The README file is also helpful. Some notes on `dlc`:

  - The `dlc` program runs silently unless it detects a problem.
  - Don't include `<stdio.h>` in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages.
- Check the file `README` for documentation on running the `btest` program. You'll find it helpful to work through the functions one at a time, testing each one as you go. You can use the `-f` flag to instruct `btest` to test only a single function, e.g., `./btest -f getByte`.

Hand In Instructions

- Make sure you have included your identifying information in your file `bits.c`.
- Write a short `readme` text file that contains your identifying information (your name(s) and your Unix login name(s)), the Unix account and directory where the file `bits.c` is located, plus any bugs and deviations from the lab specifications.
- To hand in your `readme` and your `bits.c` files when you are done, simply make sure they are in the `datalab` directory in your home directory, and then send an email to your instructor. Make sure to mention in your email the names of all members of your team so that proper credit can be given.

GOOD LUCK!